4 Creating a website with Razor Pages

This chapter covers

- Introducing Razor Pages and the Model-View-Controller (MVC) design pattern
- Using Razor Pages in ASP.NET Core
- Choosing between Razor Pages and MVC controllers
- Controlling application flow using action results

In chapter 3 you learned about the middleware pipeline, which defines how an ASP.NET Core application responds to a request. Each piece of middleware can modify or handle an incoming request before passing the request to the next middleware in the pipeline.

In ASP.NET Core web applications, your middleware pipeline will normally include the `EndpointMiddleware`. This is typically where you write the bulk of your application logic, calling various other classes in your app. It also serves as the main entry point for users to interact with your app. It typically takes one of three forms:

- *An HTML web application designed for direct use by users*—If the application is consumed directly by users, as in a traditional web application, then Razor Pages is responsible for generating the web pages that the user interacts with.

It handles requests for URLs, it receives data posted using forms, and it generates the HTML that users use to view and navigate your app.

- *An API designed for consumption by another machine or in code*—The other main possibility for a web application is to serve as an API to backend server processes, to a mobile app, or to a client framework for building single-page applications (SPAs). In this case, your application serves data in machine-readable formats such as JSON or XML instead of the human-focused HTML output.
- *Both an HTML web application and an API*—It is also possible to have applications that serve both needs. This can let you cater to a wider range of clients while sharing logic in your application.

In this chapter you'll learn how ASP.NET Core uses Razor Pages to handle the first of these options, creating server-side rendered HTML pages. You'll start by looking at the Model-View-Controller (MVC) design pattern to see the benefits that can be achieved through its use and learn why it's been adopted by so many web frameworks as a model for building maintainable applications.

Next you'll learn how the MVC design pattern applies to ASP.NET Core. The MVC pattern is a broad concept that can be applied in a variety of situations, but the use case in ASP.NET Core is specifically as a UI abstraction. You'll see how Razor Pages implements the MVC design pattern, how it builds on top of the ASP.NET Core MVC framework, and compare the two approaches.

Next you'll see how to add Razor Pages to an existing application, and how to create your first Razor Pages. You'll learn how to define page handlers to execute when your application receives a request and how to generate a result that can be used to create an HTTP response to return.

I won't cover how to create Web APIs in this chapter. Web APIs still use the ASP.NET Core MVC framework, but they're used in a slightly different way to Razor Pages. Instead of returning web pages that are directly displayed in a user's browser, they return data formatted for consumption in code. Web APIs are often used for providing data to mobile and web applications, or to other server applications. But they still follow the same general MVC pattern. You'll see how to create a Web API in chapter 9.

> **NOTE** This chapter is the first of several on Razor Pages and MVC in ASP.NET Core. As I've already mentioned, these frameworks are often responsible for handling all the business logic and UI code for your application, so, perhaps unsurprisingly, they're large and somewhat complicated. The next five chapters all deal with a different aspect of the MVC pattern that makes up the MVC and Razor Pages frameworks.

In this chapter I'll try to prepare you for each of the upcoming topics, but you may find that some of the behavior feels a bit like magic at this stage. Try not to become too concerned with exactly how all the pieces tie together; focus on the specific concepts being addressed. It should all become clear as we cover the associated details in the remainder of this first part of the book.

4.1 An introduction to Razor Pages

The Razor Pages programming model was introduced in ASP.NET Core 2.0 as a way to build server-side rendered "page-based" websites. It builds on top of the ASP.NET Core infrastructure to provide a streamlined experience, using conventions where possible to reduce the amount of boilerplate code and configuration required.

> **DEFINITION** A *page-based* website is one in which the user browses between multiple pages, enters data into forms, and generally consumes content. This contrasts with applications like games or single-page applications (SPAs), which are heavily interactive on the client side.

You've already seen a very basic example of a Razor Page in chapter 2. In this section we'll start by looking at a slightly more complex Razor Page, to better understand the overall design of Razor Pages. I will cover

- An example of a typical Razor Page
- The MVC design pattern and how it applies to Razor Pages
- How to add Razor Pages to your application

At the end of this section, you should have a good understanding of the overall design behind Razor Pages and how they relate to the MVC pattern.

4.1.1 Exploring a typical Razor Page

In chapter 2 we looked at a very simple Razor Page. It didn't contain any logic and instead just rendered the associated Razor view. This pattern may be common if you're building a content-heavy marketing website, for example, but more commonly your Razor Pages will contain some logic, load data from a database, or use forms to allow users to submit information.

To give you more of a flavor of how typical Razor Pages work, in this section we'll look briefly at a slightly more complex Razor Page. This page is taken from a to-do list application and is used to display all the to-do items for a given category. We're not focusing on the HTML generation at this point, so the following listing shows only the `PageModel` code-behind for the Razor Page.

Listing 4.1 A Razor Page for viewing all to-do items in a given category

```
public class CategoryModel : PageModel
{
    private readonly ToDoService _service;
    public CategoryModel(ToDoService service)
    {
        _service = service;
    }

    public ActionResult OnGet(string category)
    {
        Items = _service.GetItemsForCategory(category);
```

The ToDoService is provided in the model constructor using dependency injection.

The OnGet handler takes a parameter, category.

The handler calls out to the ToDoService to retrieve data and sets the Items property.

```
        return Page();
    }

    public List<ToDoListModel> Items { get; set; }
}
```

Returns a PageResult indicating the Razor view should be rendered

The Razor View can access the Items property when it is rendered.

This example is still relatively simple, but it demonstrates a variety of features compared to the basic example from chapter 2:

- The page handler, `OnGet`, accepts a method parameter, `category`. This parameter is automatically populated by the Razor Page infrastructure using values from the incoming request in a process called *model binding*. I discuss model binding in detail in chapter 6.
- The handler doesn't interact with the database directly. Instead, it uses the `category` value provided to interact with the `ToDoService`, which is injected as a constructor argument using dependency injection.
- The handler returns `Page()` at the end of the method to indicate that the associated Razor view should be rendered. The `return` statement is actually optional in this case; by convention, if the page handler is a `void` method, the Razor view will still be rendered, behaving as if you had called `return Page()` at the end of the method.
- The Razor View has access to the `CategoryModel` instance, so it can access the `Items` property that is set by the handler. It uses these items to build the HTML that is ultimately sent to the user.

The pattern of interactions in the Razor Page of listing 4.1 shows a common pattern. The page handler is the central controller for the Razor Page. It receives an input from the user (the `category` method parameter), calls out to the "brains" of the application (the `ToDoService`) and passes data (by exposing the `Items` property) to the Razor view, which generates the HTML response. If you squint, this looks like the Model-View-Controller (MVC) design pattern.

Depending on your background in software development, you may have previously come across the MVC pattern in some form. In web development, MVC is a common paradigm and is used in frameworks such as Django, Rails, and Spring MVC. But as it's such a broad concept, you can find MVC in everything from mobile apps to rich-client desktop applications. Hopefully that's indicative of the benefits the pattern can bring if used correctly! In the next section we'll look at the MVC pattern in general and how it's used by ASP.NET Core.

4.1.2 The MVC design pattern

The MVC design pattern is a common pattern for designing apps that have UIs. The original MVC pattern has many different interpretations, each of which focuses on a slightly different aspect of the pattern. For example, the original MVC design pattern was specified with rich-client graphical user interface (GUI) apps in mind, rather than

web applications, so it uses terminology and paradigms associated with a GUI environment. Fundamentally, though, the pattern aims to separate the management and manipulation of data from its visual representation.

Before I dive too far into the design pattern itself, let's consider a typical request. Imagine that a user of your application requests the Razor Page from the previous section that displays a to-do list category. Figure 4.1 shows how a Razor Page handles different aspects of a request, all of which combine to generate the final response.

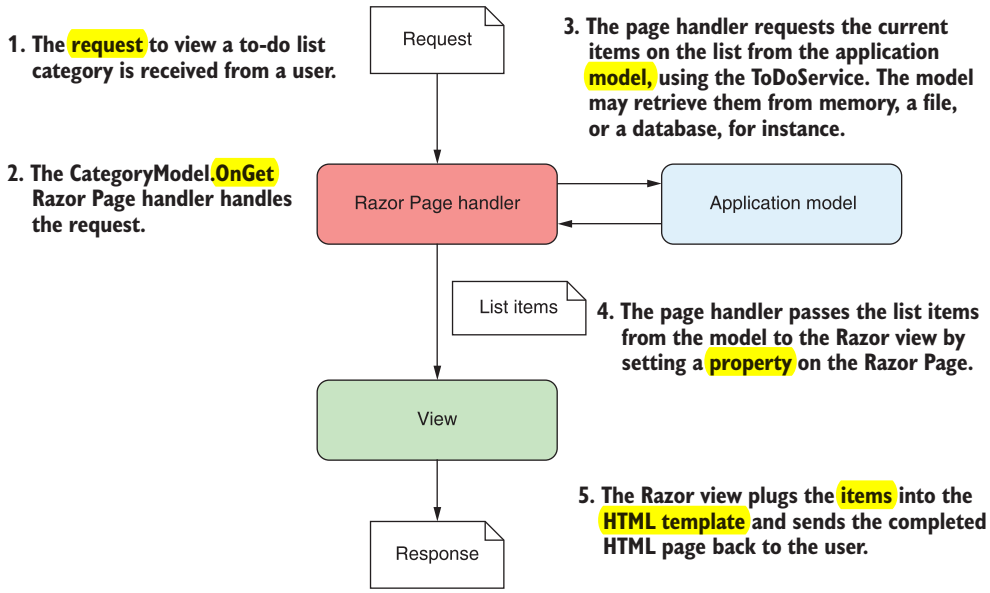

Figure 4.1 Requesting a to-do list page for a Razor Pages application. A different "component" handles each aspect of the request.

In general, three "components" make up the MVC design pattern:

- *Model*—This is the data that needs to be displayed, the global state of the application. It's accessed via the `ToDoService` in listing 4.1.
- *View*—The template that displays the data provided by the model.
- *Controller*—This updates the model and provides the data for display to the view. This role is taken by the page handler in Razor Pages. This is the `OnGet` method in listing 4.1.

Each component in the MVC design pattern is responsible for a single aspect of the overall system, which, when combined, can be used to generate a UI. The to-do list example considers MVC in terms of a web application using Razor Pages, but a request could also be equivalent to the click of a button in a desktop GUI application.

In general, the order of events when an application responds to a user interaction or request is as follows:

1. The controller (the Razor Page handler) receives the request.
2. Depending on the request, the controller either fetches the requested data from the application model using injected services, or it updates the data that makes up the model.
3. The controller selects a view to display and passes a representation of the model to it.
4. The view uses the data contained in the model to generate the UI.

When we describe MVC in this format, the controller (the Razor Page handler) serves as the entry point for the interaction. The user communicates with the controller to instigate an interaction. In web applications, this interaction takes the form of an HTTP request, so when a request to a URL is received, the controller handles it.

Depending on the nature of the request, the controller may take a variety of actions, but the key point is that the actions are undertaken using the application model. The model here contains all the business logic for the application, so it's able to provide requested data or perform actions.

> **NOTE** In this description of MVC, the model is considered to be a complex beast, containing all the logic for how to perform an action, as well as any internal state. The Razor Page `PageModel` class is *not* the model we're talking about! Unfortunately, as in all software development, naming things is hard.

Consider a request to view a product page for an e-commerce application. The controller would receive the request and would know how to contact some product service that's part of the application model. This might fetch the details of the requested product from a database and return them to the controller.

Alternatively, imagine that a controller receives a request to add a product to the user's shopping cart. The controller would receive the request and most likely would invoke a method on the model to request that the product be added. The model would then update its internal representation of the user's cart, by adding, for example, a new row to a database table holding the user's data.

> **TIP** You can think of each Razor Page handler as a mini controller focused on a single page. Every web request is another independent call to a controller that orchestrates the response. Although there are many different controllers, the handlers all interact with the *same* application model.

After the model has been updated, the controller needs to decide what response to generate. One of the advantages of using the MVC design pattern is that the model representing the application's data is decoupled from the final representation of that data, called the *view*. The controller is responsible for deciding whether the response should generate an HTML view, whether it should send the user to a new page, or whether it should return an error page.

One of the advantages of the model being independent of the view is that it improves testability. UI code is classically hard to test, as it's dependent on the environment—anyone who has written UI tests simulating a user clicking buttons and typing in forms knows that it's typically fragile. By keeping the model independent of the view, you can ensure the model stays easily testable, without any dependencies on UI constructs. As the model often contains your application's business logic, this is clearly a good thing!

The view can use the data passed to it by the controller to generate the appropriate HTML response. The view is only responsible for generating the final representation of the data; it's not involved in any of the business logic.

This is all there is to the MVC design pattern in relation to web applications. Much of the confusion related to MVC seems to stem from slightly different uses of the term for slightly different frameworks and types of applications. In the next section, I'll show how the ASP.NET Core framework uses the MVC pattern with Razor Pages, along with more examples of the pattern in action.

4.1.3 Applying the MVC design pattern to Razor Pages

In the previous section I discussed the MVC *pattern* as it's typically used in web applications; Razor Pages use this pattern. But ASP.NET Core also includes a *framework* called ASP.NET Core MVC. This framework (unsurprisingly) very closely mirrors the MVC design pattern, using *controllers* and *action methods* in place of Razor Pages and page handlers. Razor Pages builds directly on top of the underlying ASP.NET Core MVC framework, using the MVC framework under the hood for their behavior.

If you prefer, you can avoid Razor Pages entirely and work with the MVC framework directly in ASP.NET Core. This was the *only* option in early versions of ASP.NET Core and the previous version of ASP.NET.

> **TIP** I look in greater depth at choosing between Razor Pages and the MVC framework in section 4.2.

In this section we'll look in greater depth at how the MVC design pattern applies to Razor Pages in ASP.NET Core. This will also help clarify the role of various features of Razor Pages.

Do Razor Pages use MVC or MVVM?

Occasionally I've seen people describe Razor Pages as using the Model-View-View Model (MVVM) design pattern, rather than the MVC design pattern. Personally, I don't agree, but it's worth being aware of the differences.

MVVM is a UI pattern that is often used in mobile apps, desktop apps, and some client-side frameworks. It differs from MVC in that there is a bidirectional interaction between the view and the view model. The view model tells the view what to display, but the view can also trigger changes directly on the view model. It's often used with two-way data binding where a view model is "bound" to a view.

(continued)

Some people consider the Razor Pages `PageModel` to be filling this role, but I'm not convinced. Razor Pages definitely seems based on the MVC pattern to me (it's based on the ASP.NET Core MVC framework after all!), and it doesn't have the same two-way binding that I would expect with MVVM.

As you've seen in previous chapters, ASP.NET Core implements Razor Page endpoints using a combination of `RoutingMiddleware` and `EndpointMiddleware`, as shown in figure 4.2. Once a request has been processed by earlier middleware (and assuming none of them has handled the request and short-circuited the pipeline), the routing middleware will select which Razor Page handler should be executed, and the endpoint middleware executes the page handler.

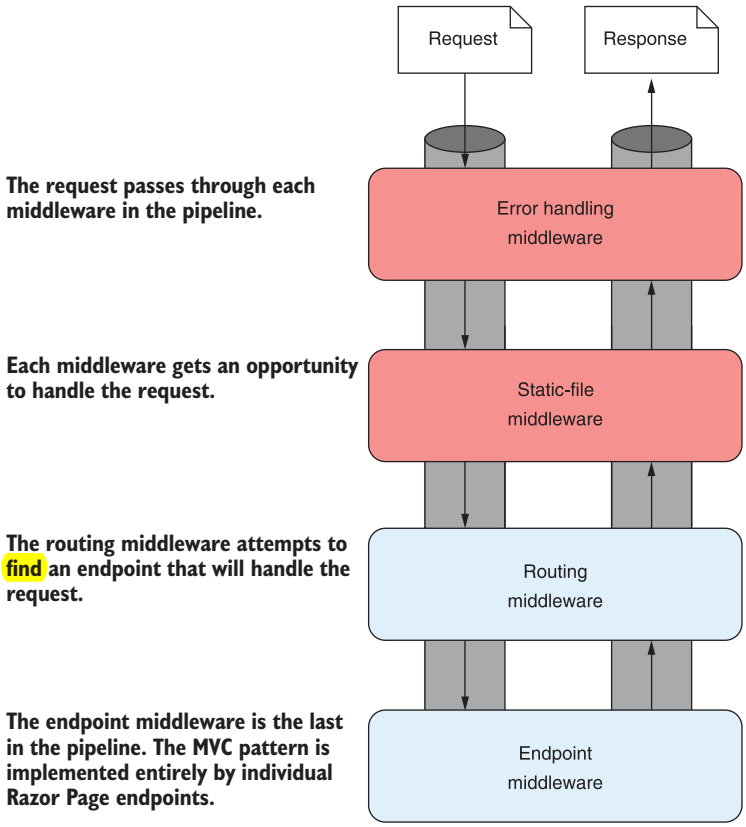

Figure 4.2 The middleware pipeline for a typical ASP.NET Core application. The request is processed by each middleware in sequence. If the request reaches the routing middleware, the middleware selects an endpoint, such as a Razor Page, to execute. The endpoint middleware executes the selected endpoint.

Middleware often handles cross-cutting concerns or narrowly defined requests, such as requests for files. For requirements that fall outside of these functions, or that have many external dependencies, a more robust framework is required. Razor Pages (and/or ASP.NET Core MVC) can provide this framework, allowing interaction with your application's core business logic and the generation of a UI. It handles everything from mapping the request to an appropriate controller to generating the HTML or API response.

In the traditional description of the MVC design pattern, there's only a single type of model, which holds all the non-UI data and behavior. The controller updates this model as appropriate and then passes it to the view, which uses it to generate a UI.

One of the problems when discussing MVC is the vague and ambiguous terms that it uses, such as "controller" and "model." Model, in particular, is such an overloaded term that it's often difficult to be sure exactly what it refers to—is it an object, a collection of objects, an abstract concept? Even ASP.NET Core uses the word "model" to describe several related, but different, components, as you'll see shortly.

Directing a request to a Razor Page and building a binding model

The first step when your app receives a request is routing the request to an appropriate Razor Page handler. Let's think about the category to-do list page again, from listing 4.1. On this page, you're displaying a list of items that have a given category label. If you're looking at the list of items with a category of "Simple," you'd make a request to the `/category/Simple` path.

Routing takes the headers and path of the request, `/category/Simple`, and maps it against a preregistered list of patterns. These patterns each match a path to a single Razor Page and page handler. You'll learn more about routing in the next chapter.

> **TIP** I'm using the term *Razor Page* to refer to the combination of the Razor view and the `PageModel` that includes the page handler. Note that that `PageModel` class is *not* the "model" we're referring to when describing the MVC pattern. It fulfills other roles, as you will see later in this section.

Once a page handler is selected, the *binding model* (if applicable) is generated. This model is built based on the incoming request, the properties of the `PageModel` marked for binding, and the method parameters required by the page handler, as shown in figure 4.3. A binding model is normally one or more standard C# objects, with properties that map to the requested data. We'll look at binding models in detail in chapter 6.

> **DEFINITION** A *binding model* is one or more objects that act as a "container" for the data provided in a request—data that's required by a page handler.

In this case, the binding model is a simple string, `category`, which is bound to the `"Simple"` value. This value is provided in the request URL's path. A more complex binding model could also have been used, where multiple properties were populated.

The binding model in this case corresponds to the method parameter of the `OnGet` page handler. An instance of the Razor Page is created using its constructor, and the

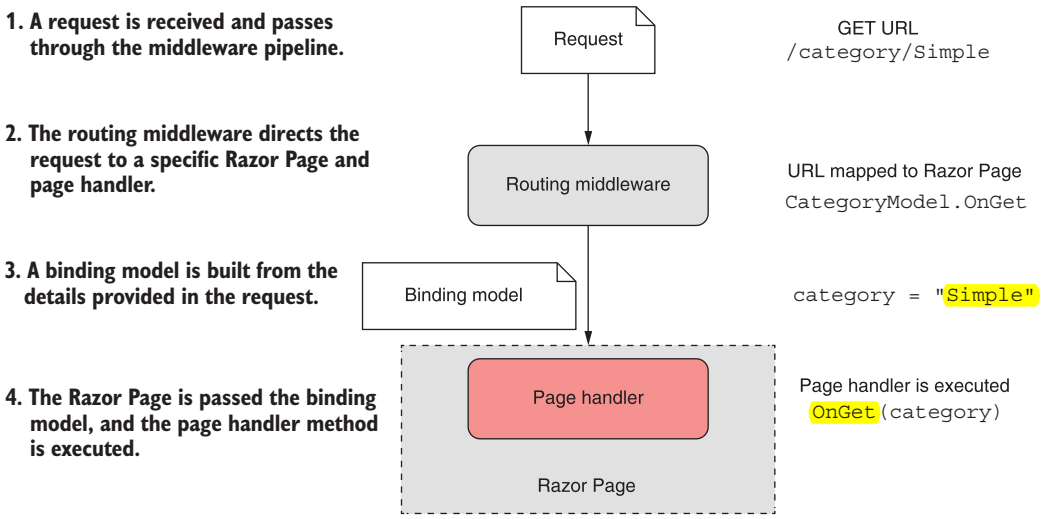

Figure 4.3 Routing a request to a controller and building a binding model. A request to the `/category/Simple` URL results in the `CategoryModel.OnGet` page handler being executed, passing in a populated binding model, `category`.

binding model is passed to the page handler when it executes, so it can be used to decide how to respond. For this example, the page handler uses it to decide which to-do items to display on the page.

EXECUTING A HANDLER USING THE APPLICATION MODEL

The role of the page handler as the controller in the MVC pattern is to *coordinate* the generation of a response to the request it's handling. That means it should perform only a limited number of actions. In particular, it should

- Validate that the data contained in the binding model provided is valid for the request
- Invoke the appropriate actions on the application model using services
- Select an appropriate response to generate based on the response from the application model

Figure 4.4 shows the page handler invoking an appropriate method on the application model. Here, you can see that the application model is a somewhat abstract concept that encapsulates the remaining non-UI parts of your application. It contains the *domain model*, a number of services, and the database interaction.

> **DEFINITION** The *domain model* encapsulates complex business logic in a series of classes that don't depend on any infrastructure and can be easily tested.

The page handler typically calls into a single point in the application model. In our example of viewing a to-do list category, the application model might use a variety of

1. The page handler uses the category provided in the binding model to determine which method to invoke in the application model.
2. The page handler method calls into services that make up the application model. This might use the domain model to determine whether to include completed to-do items, for example.
3. The services load the details of the to-do items from the database and return them to the action method.

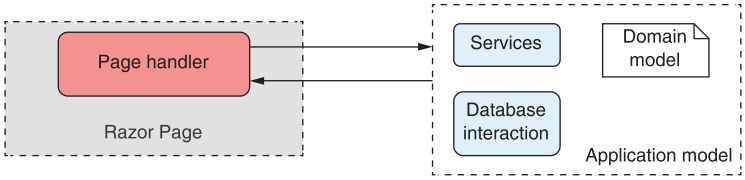

Figure 4.4 When executed, an action will invoke the appropriate methods in the application model.

services to check whether the current user is allowed to view certain items, to search for items in the given category, to load the details from the database, or to load a picture associated with an item from a file.

Assuming the request is valid, the application model will return the required details to the page handler. It's then up to the page handler to choose a response to generate.

Building HTML using the view model

Once the page handler has called out to the application model that contains the application business logic, it's time to generate a response. A *view model* captures the details necessary for the view to generate a response.

> **DEFINITION** A *view model* in the MVC pattern is all the data required by the view to render a UI. It's typically some transformation of the data contained in the application model, plus extra information required to render the page, such as the page's title.

The term *view model* is used extensively in ASP.NET Core MVC, where it typically refers to a single object that is passed to the Razor view to render. However, with Razor Pages, the Razor view can access the Razor Page's *page model* class directly. Therefore, the Razor Page `PageModel` typically *acts* as the view model in Razor Pages, with the data required by the Razor view exposed via properties, as you saw previously in listing 4.1.

> **NOTE** Razor Pages use the `PageModel` class itself as the view model for the Razor view, by exposing the required data as properties.

The Razor view uses the data exposed in the page model to generate the final HTML response. Finally, this is sent back through the middleware pipeline and out to the user's browser, as shown in figure 4.5.

It's important to note that although the page handler selects *whether* to execute the view and the data to use, it doesn't control *what HTML is generated*. It's the view itself that decides what the content of the response will be.

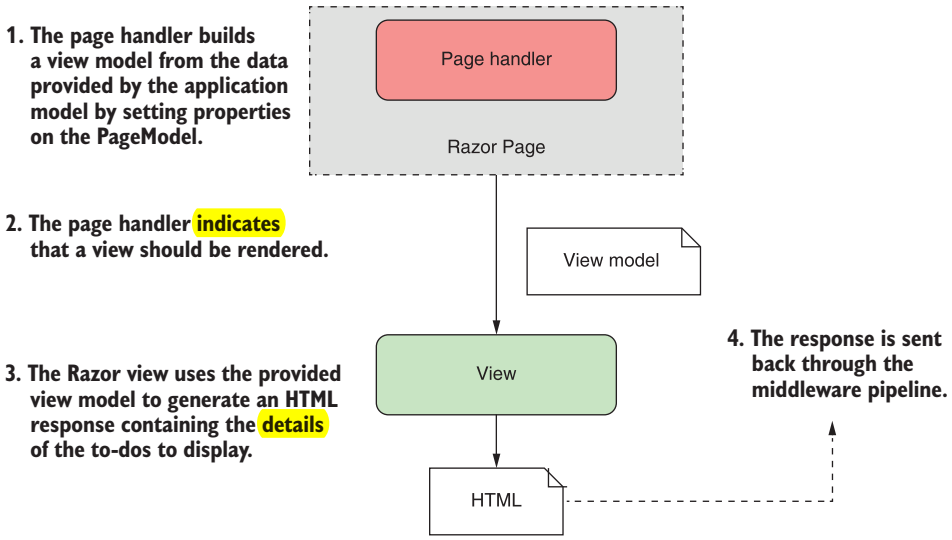

Figure 4.5 The page handler builds a view model by setting properties on the `PageModel`. It's the view that generates the response.

PUTTING IT ALL TOGETHER: A COMPLETE RAZOR PAGE REQUEST

Now that you've seen each of the steps that goes into handling a request in ASP.NET Core using Razor Pages, let's put it all together from request to response. Figure 4.6 shows how the steps combine to handle the request to display the list of to-do items for the "Simple" category. The traditional MVC pattern is still visible in Razor Pages, made up of the page handler (controller), the view, and the application model.

By now, you might be thinking this whole process seems rather convoluted—so many steps to display some HTML! Why not allow the application model to create the view directly, rather than having to go on a dance back and forth with the page handler method?

The key benefit throughout this process is the *separation of concerns*:

- The view is responsible only for taking some data and generating HTML.
- The application model is responsible only for executing the required business logic.
- The page handler (controller) is responsible only for validating the incoming request and selecting which response is required, based on the output of the application model.

By having clearly defined boundaries, it's easier to update and test each of the components without depending on any of the others. If your UI logic changes, you won't necessarily have to modify any of your business logic classes, so you're less likely to introduce errors in unexpected places.

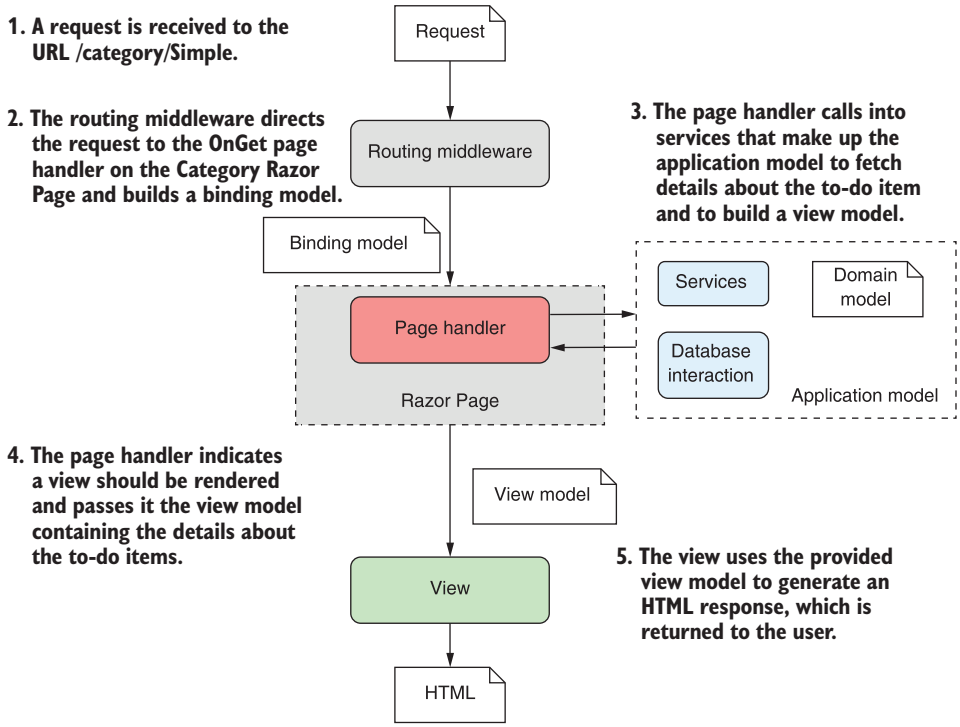

Figure 4.6 A complete Razor Pages request for the list of to-dos in the "Simple" category

The dangers of tight coupling

Generally speaking, it's a good idea to reduce coupling between logically separate parts of your application as much as possible. This makes it easier to update your application without causing adverse effects or requiring modifications in seemingly unrelated areas. Applying the MVC pattern is one way to help with this goal.

As an example of when coupling rears its head, I remember a case a few years ago when I was working on a small web app. In our haste, we had not properly decoupled our business logic from our HTML generation code, but initially there were no obvious problems—the code worked, so we shipped it!

A few months later, someone new started working on the app and immediately "helped" by renaming an innocuous spelling error in a class in the business layer. Unfortunately, the names of those classes had been used to generate our HTML code, so renaming the class caused the whole website to break in users' browsers! Suffice it to say, we made a concerted effort to apply the MVC pattern after that and ensure that we had a proper separation of concerns.

The examples shown in this chapter demonstrate the bulk of the Razor Pages functionality. It has additional features, such as the filter pipeline, which I'll cover later

(chapter 13), and I'll discuss binding models in greater depth in chapter 6, but the overall behavior of the system is the same.

Similarly, in chapter 9 I'll discuss how the MVC design pattern applies when you're generating machine-readable responses using Web API controllers. The process is, for all intents and purposes, identical, apart from the final result generated.

In the next section, you'll see how to add Razor Pages to your application. Some templates in Visual Studio and the .NET CLI will include Razor Pages by default, but you'll see how to add it to an existing application and explore the various options available.

4.1.4 Adding Razor Pages to your application

The MVC infrastructure, whether used by Razor Pages or MVC/API controllers, is a foundational aspect of all but the simplest ASP.NET Core applications, so virtually all templates include it configured by default in some way. But to make sure you're comfortable with adding Razor Pages to an existing project, I'll show you how to start with a basic empty application and add Razor Pages to it from scratch.

The result of your efforts won't be exciting yet. We'll display "Hello World" on a web page, but this will show how simple it is to convert an ASP.NET Core application to use Razor Pages. It will also emphasize the pluggable nature of ASP.NET Core—if you don't need the functionality provided by Razor Pages, you don't have to use it.

Here's how you add Razor Pages to your application:

1. In Visual Studio 2019, choose File > New > Project or choose Create a New Project from the splash screen.
2. From the list of templates, choose ASP.NET Core Web Application, ensuring you select the C# language template.
3. On the next screen, enter a project name, location, and solution name, and click Create.
4. On the following screen, create a basic template without MVC or Razor Pages by selecting the ASP.NET Core Empty project template in Visual Studio, as shown in figure 4.7. You can create a similar empty project using the .NET CLI with the `dotnet new web` command.
5. Add the necessary Razor Page services (shown in bold) in your Startup.cs file's `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
}
```

6. Replace the existing basic endpoint configured in the `EndpointMiddleware` at the end of your middleware pipeline with the `MapRazorPages()` extension method

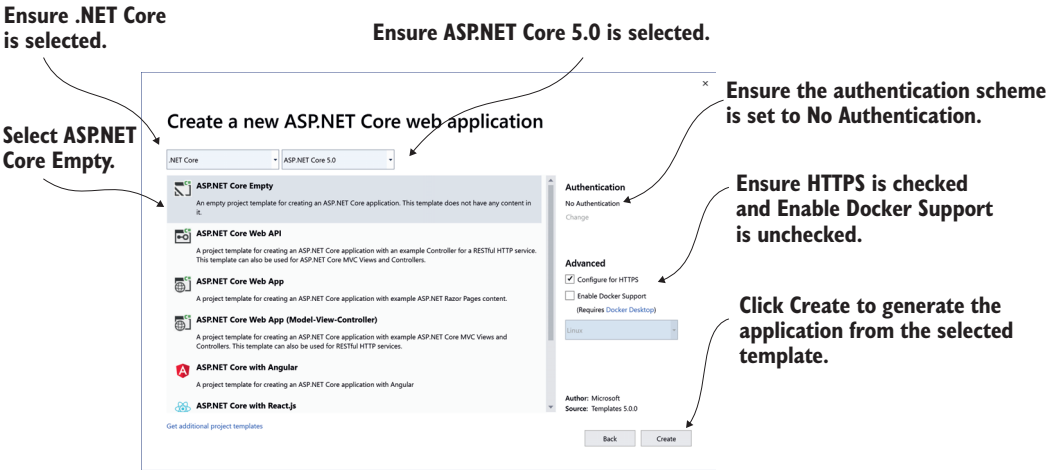

Figure 4.7 Creating an empty ASP.NET Core template. The empty template will create a simple ASP.NET Core application that contains a small middleware pipeline without Razor Pages.

(in bold). For simplicity, also remove the existing error handler middleware from the `Configure` method of Startup.cs for now:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

7 Right-click your project in Solution Explorer and choose Add > New Folder to add a new folder to the root of your project. Name the new folder "Pages".

You have now configured your project to use Razor Pages, but you don't have any pages yet. The following steps add a new Razor Page to your application. You can create a similar Razor Page using the .NET CLI by running `dotnet new page -n Index -o Pages/` from the project directory.

8 Right-click the new pages folder and choose Add > Razor Page, as shown in figure 4.8.

9 On the following page, select Razor Page – Empty and click Add. In the following dialog box, name your page Index.cshtml, as shown in figure 4.9.

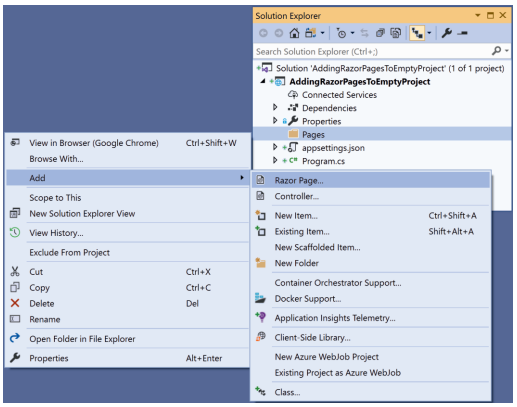

Figure 4.8 Adding a new Razor Page to your project

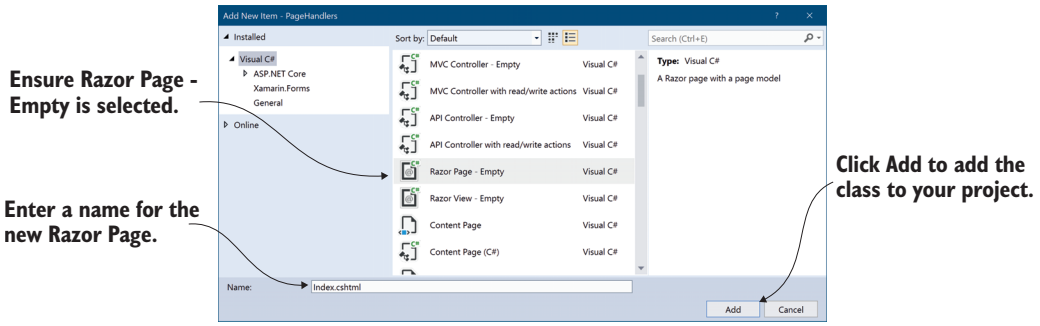

Figure 4.9 Creating a new Razor Page using the Add Razor Page dialog box

10 After Visual Studio has finished generating the file, open the Index.cshtml file, and update the HTML to say `Hello World!` by replacing the file contents with the following:

```
@page
@model AddingRazorPagesToEmptyProject.IndexModel
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
```

```
<body>
    <h1>Hello World!</h1>
</body>
</html>
```

Once you've completed all these steps, you should be able to restore, build, and run your application.

> **NOTE** You can run your project by pressing F5 from within Visual Studio (or by calling `dotnet run` at the command line from the project folder). This will restore any referenced NuGet packages, build your project, and start your application. Visual Studio will automatically open a browser window to access your application's home page.

When you make a request to the root `"/"` path, the application invokes the `OnGet` handler on the `IndexModel` due to the conventional way routing works for Razor Pages based on the filename. Don't worry about this for now; we'll go into it in detail in the next chapter.

The `OnGet` handler is a `void` method, which causes the Razor Page to render the associated Razor view and send it to the user's browser in the response.

Razor Pages rely on a number of internal services to perform their functions, which must be registered during application startup. This is achieved with the call to `AddRazorPages` in the `ConfigureServices` method of Startup.cs. Without this, you'll get exceptions when your app starts, reminding you that the call is required.

The call to `MapRazorPages` in `Configure` registers the Razor Page endpoints with the endpoint middleware. As part of this call, the routes that are used to map URL paths to specific Razor Page handlers are registered automatically.

> **NOTE** I cover routing in detail in the next chapter.

The instructions in this section described how to add Razor Pages to your application, but that's not the only way to add HTML generation to your application. As I mentioned previously, Razor Pages builds on top of the ASP.NET Core MVC framework and shares many of the same concepts. In the next section, we'll take a brief look at MVC controllers, see how they compare to Razor Pages, and discuss when you should choose to use one approach over the other.

4.2 Razor Pages vs. MVC in ASP.NET Core

In this book, I focus on Razor Pages, as that has become the recommended approach for building server-side rendered applications with ASP.NET Core. However, I also mentioned that Razor Pages uses the ASP.NET Core MVC framework behind the scenes, and that you can choose to use it directly if you wish. Additionally, if you're creating a Web API for working with mobile or client-side apps, you will almost certainly be using the MVC framework directly.

NOTE I look at how to build Web APIs in chapter 9.

So what are the differences between Razor Pages and MVC, and when should you choose one or the other?

If you're new to ASP.NET Core, the answer is pretty simple—use Razor Pages for server-side rendered applications, and use the MVC framework for building Web APIs. There are nuances I'll discuss in later sections, but that distinction will serve you very well initially.

If you're familiar with the previous version of ASP.NET or earlier versions of ASP.NET Core and are deciding whether to use Razor Pages, then this section should help you choose. Developers coming from those backgrounds often have misconceptions about Razor Pages initially (as I did!), incorrectly equating them to Web Forms and overlooking their underlying basis of the MVC framework.

Before we can get to comparisons, though, we should take a brief look at the ASP.NET Core MVC framework itself. Understanding the similarities and differences between MVC and Razor Pages can be very useful, as you'll likely find a use for MVC at some point, even if you use Razor Pages most of the time.

4.2.1 MVC controllers in ASP.NET Core

In section 4.1 we looked at the MVC design pattern, and at how it applies to Razor Pages in ASP.NET Core. Perhaps unsurprisingly, you can use the ASP.NET Core MVC framework in almost exactly the same way. To demonstrate the difference between Razor Pages and MVC, we'll look at an MVC version of the Razor Page from listing 4.1, which displays a list of to-do items for a given category.

Instead of a `PageModel` and page handler, MVC uses the concept of *controllers* and *action methods*. These are almost directly analogous to their Razor Pages counterparts, as you can see in figure 4.10, which shows an MVC equivalent of figure 4.6. On the other hand, MVC controllers use explicit *view model*s to pass data to a Razor view, rather than exposing the data as properties on itself (as Razor Pages do with page models).

DEFINITION An *action* (or *action method*) is a method that runs in response to a request. An *MVC controller* is a class that contains a number of logically grouped action methods.

Listing 4.2 shows an example of how an MVC controller that provides the same functionality as the Razor Page in listing 4.1 might look. In MVC, controllers are often used to aggregate similar actions together, so the controller in this case is called `ToDoController`, as it would typically contain additional action methods for working with to-do items, such as actions to view a specific item, or to create a new one.

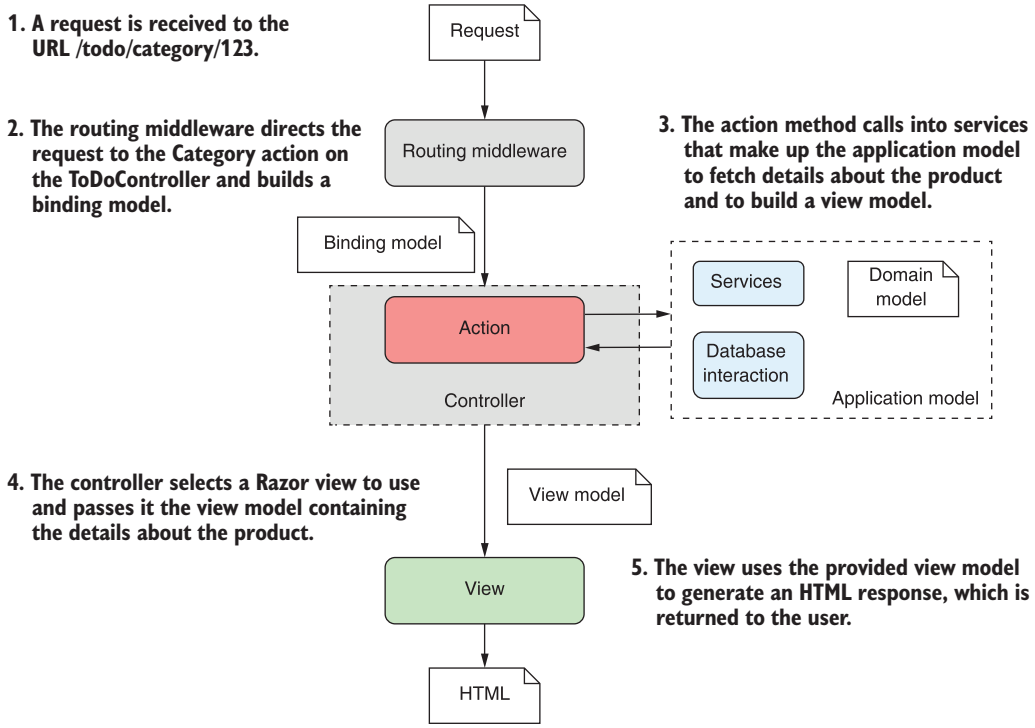

Figure 4.10 A complete MVC controller request for a category. The MVC controller pattern is almost identical to that of Razor Pages, shown in figure 4.6. The controller is equivalent to a Razor Page, and the action is equivalent to a page handler.

Listing 4.2 An MVC controller for viewing all to-do items in a given category

```
public class ToDoController : Controller
{
    private readonly ToDoService _service;
    public ToDoController(ToDoService service)
    {
        _service = service;
    }

    public ActionResult Category(string id)
    {
        var items = _service.GetItemsForCategory(id);
        var viewModel = new CategoryViewModel(items);

        return View(viewModel);
    }
```

The ToDoService is provided in the controller constructor using dependency injection.

The Category action method takes a parameter, id.

The action method calls out to the ToDoService to retrieve data and build a view model.

The view model is a simple C# class, defined elsewhere in your application.

Returns a ViewResult indicating the Razor view should be rendered, passing in the view model

```
    public ActionResult Create(ToDoListModel model)
    {
        // ...
    }
}
```

MVC controllers often contain multiple action methods that respond to different requests.

Aside from some naming differences, the `ToDoController` looks very similar to the Razor Page equivalent from listing 4.1. Indeed, *architecturally*, Razor Pages and MVC are essentially equivalent, as they both use the MVC design pattern. The most obvious differences relate to *where the files are placed* in your project, as I discuss in the next section.

4.2.2 The benefits of Razor Pages

In the previous section I showed that the code for an MVC controller looks very similar to the code for a Razor Page `PageModel`. If that's the case, what benefit is there to using Razor Pages? In this section I discuss some of the pain points of MVC controllers and how Razor Pages attempts to address them.

Razor Pages are not Web Forms

A common argument I hear from existing ASP.NET developers against Razor Pages is "oh, they're just Web Forms." That sentiment misses the mark in many different ways, but it's common enough that it's worth addressing directly.

Web Forms was a web-programming model that was released as part of .NET Framework 1.0 in 2002. They attempted to provide a highly productive experience for developers moving from desktop development to the web for the first time.

Web Forms are much maligned now, but their weakness only became apparent later. Web Forms attempted to hide the complexities of the web away from you, to give you the impression of developing with a desktop app. That often resulted in apps that were slow, with lots of interdependencies, and that were hard to maintain.

Web Forms provided a page-based programming model, which is why Razor Pages sometimes gets associated with them. However, as you've seen, Razor Pages is based on the MVC design pattern, and it exposes the intrinsic features of the web without trying to hide them from you.

Razor Pages optimizes certain flows using conventions (some of which you've seen), but it's not trying to build a *stateful* application model over the top of a stateless web application, in the way that Web Forms did.

In MVC, a single controller can have multiple action methods. Each action handles a different request and generates a different response. The grouping of multiple actions in a controller is somewhat arbitrary, but it's typically used to group actions related to a specific entity: to-do list items in this case. A more complete version of the `ToDoController` in listing 4.2 might include action methods for listing all to-do

items, for creating new items, and for deleting items, for example. Unfortunately, you can often find that your controllers become very large and bloated, with many dependencies.[1]

> **NOTE** You don't *have* to make your controllers very large like this. It's just a common pattern. You could, for example, create a separate controller for every action instead.

Another pitfall of the MVC controllers is the way they're typically organized in your project. Most action methods in a controller will need an associated Razor view, and a view model for passing data to the view. The MVC approach traditionally groups classes by *type* (controller, view, view model), while the Razor Page approach groups by *function*—everything related to a specific page is co-located.

Figure 4.11 compares the file layout for a simple Razor Pages project with the MVC equivalent. Using Razor Pages means much less scrolling up and down between the controller, views, and view model folders whenever you're working on a particular page. Everything you need is found in two files, the .cshtml Razor view and the .cshtml.cs `PageModel` file.

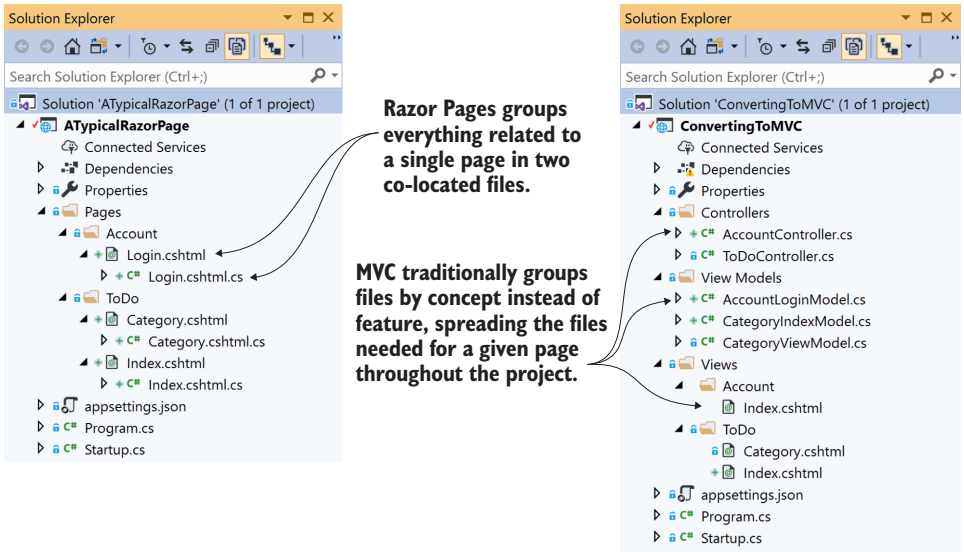

Figure 4.11 Comparing the folder structure for an MVC project to the folder structure for a Razor Pages project

[1] Before moving to Razor Pages, the ASP.NET Core template that includes user login functionality contained two such controllers, each containing over 20 action methods and over 500 lines of code!

There are additional differences between MVC and Razor Pages, which I'll highlight throughout the book, but this layout difference is really the biggest win. Razor Pages embraces the fact that you're building a page-based application and optimizes your workflow by keeping everything related to a single page together.

> **TIP** You can think of each Razor Page as a mini controller focused on a single page. Page handlers are functionally equivalent to MVC controller action methods.

This layout also has the benefit of making each page a separate class. This contrasts with the MVC approach of making each page an *action* on a given controller. Each Razor Page is cohesive for a particular *feature*, such as displaying a to-do item. MVC controllers contain action methods that handle multiple different features for a more abstract *concept*, such as all the features related to to-do items.

Another important point is that Razor Pages doesn't lose any of the separation-of-concerns that MVC has. The view part of Razor Pages is still only concerned with rendering HTML, and the handler is the coordinator that calls out to the application model. The only real difference is the lack of the explicit view model that you have in MVC, but it's perfectly possible to emulate this in Razor Pages if that's a deal breaker for you.

The benefits of using Razor Pages are particularly noticeable when you have "content" websites, such as marketing websites, where you're mostly displaying static data, and there's no real logic. In that case, MVC adds complexity without any real benefits, as there's not really any logic in the controllers at all. Another great use case is when you're creating forms for users to submit data. Razor Pages is especially optimized for this scenario, as you'll see in later chapters.

Clearly, I'm a fan of Razor Pages, but that's not to say they're perfect for every situation. In the next section I discuss some of the cases when you might choose to use MVC controllers in your application. Bear in mind it's not an either-or choice—it's possible to use both MVC controllers and Razor Pages in the same application, and in many cases that may be the best option.

4.2.3 When to choose MVC controllers over Razor Pages

Razor Pages are great for building page-based server-side rendered applications. But not all applications fit that mold, and even some applications that *do* fall in that category might be best developed using MVC controllers instead of Razor Pages. These are a few such scenarios:

- *When you don't want to render views*—Razor Pages are best for page-based applications, where you're rendering a view for the user. If you're building a Web API, you should use MVC controllers instead.
- *When you're converting an existing MVC application to ASP.NET Core*—If you already have an ASP.NET application that uses MVC, it's probably not worth converting your existing MVC controllers to Razor Pages. It makes more sense to keep your

existing code, and perhaps to look at doing *new* development in the application with Razor Pages.

- *When you're doing a lot of partial page updates*—It's possible to use JavaScript in an application to avoid doing full page navigations by only updating part of the page at a time. This approach, halfway between fully server-side rendered and a client-side application may be easier to achieve with MVC controllers than Razor Pages.

When not to use Razor Pages or MVC controllers

Typically you'll use either Razor Pages or MVC controllers to write most of your application logic for an app. You'll use it to define the APIs and pages in your application and to define how they interface with your business logic. Razor Pages and MVC provide an extensive framework (as you'll see over the next six chapters) that provide a great deal of functionality to help build your apps quickly and efficiently. But they're not suited to *every* app.

Providing so much functionality necessarily comes with a certain degree of performance overhead. For typical apps, the productivity gains from using MVC or Razor Pages strongly outweigh any performance impact. But if you're building small, lightweight apps for the cloud, you might consider using custom middleware directly (see chapter 19) or an alternative protocol like gRPC (https://docs.microsoft.com/aspnet/core/grpc). You might want to also take a look at *Microservices in .NET Core* by Christian Horsdal Gammelgaard (Manning, 2017).

Alternatively, if you're building an app with real-time functionality, you'll probably want to consider using WebSockets instead of traditional HTTP requests. ASP.NET Core SignalR can be used to add real-time functionality to your app by providing an abstraction over WebSockets. SignalR also provides simple transport fallbacks and a remote procedure call (RPC) app model. For details, see the documentation at https://docs.microsoft.com/aspnet/core/signalr.

Another option available in ASP.NET Core 5.0 is Blazor. This framework allows you to build interactive client-side web applications by either leveraging the WebAssembly standard to run .NET code directly in your browser, or by using a stateful model with SignalR. See the documentation for details, at https://docs.microsoft.com/aspnet/core/blazor/.

Hopefully by this point you're sold on Razor Pages and their overall design. So far, all the Razor Pages we've looked at have used a single page handler. In the next section we'll look in greater depth at page handlers: how to define them, how to invoke them, and how to use them to render Razor views.

4.3 Razor Pages and page handlers

In the first section of this chapter, I described the MVC design pattern and how it relates to ASP.NET Core. In the design pattern, the controller receives a request and is the entry point for UI generation. For Razor Pages, the entry point is a page handler

that resides in a Razor Page's `PageModel`. A page handler is a method that runs in response to a request.

By default, the path of a Razor Page on disk controls the URL path that the Razor Page responds to. For example, a request to the URL `/products/list` corresponds to the Razor Page at the path `pages/Products/List.cshtml`. Razor Pages can contain any number of page handlers, but only one runs in response to a given request.

> **NOTE** You'll learn more about this process of selecting a Razor Page and handler, called *routing*, in the next chapter.

The responsibility of a page handler is generally threefold:

- Confirm that the incoming request is valid.
- Invoke the appropriate business logic corresponding to the incoming request.
- Choose the appropriate *kind* of response to return.

A page handler doesn't need to perform all these actions, but at the very least it must choose the kind of response to return. Page handlers typically return one of three things:

- *A* `PageResult` *object*—This causes the associated Razor view to generate an HTML response.
- *Nothing (the handler returns* `void` *or* `Task`*)*—This is the same as the previous case, causing the Razor view to generate an HTML response.
- *A* `RedirectToPageResult`—This indicates that the user should be redirected to a different page in your application.

These are the most commonly used results for Razor Pages, but I describe some additional options in section 4.3.2.

It's important to realize that a page handler doesn't generate a response directly; it selects the *type* of response and prepares the data for it. For example, returning a `PageResult` doesn't generate any HTML at that point; it merely indicates that a view *should* be rendered. This is in keeping with the MVC design pattern in which it's the *view* that generates the response, not the *controller*.

> **TIP** The page handler is responsible for choosing what sort of response to send; the *view engine* in the MVC framework uses the result to generate the response.

It's also worth bearing in mind that page handlers should generally not be performing business logic directly. Instead, they should call appropriate services in the application model to handle requests. If a page handler receives a request to add a product to a user's cart, it shouldn't directly manipulate the database or recalculate cart totals, for example. Instead, it should make a call to another class to handle the details. This approach of separating concerns ensures your code stays testable and maintainable as it grows.

4.3.1 Accepting parameters to page handlers

Some requests made to page handlers will require additional values with details about the request. If the request is for a search page, the request might contain details of the search term and the page number they're looking at. If the request is posting a form to your application, such as a user logging in with their username and password, those values must be contained in the request. In other cases, there will be no values, such as when a user requests the home page for your application.

The request may contain additional values from a variety of different sources. They could be part of the URL, the query string, headers, or the body of the request itself. The middleware will extract values from each of these sources and convert them into .NET types.

> **DEFINITION** The process of extracting values from a request and converting them to .NET types is called *model binding*. I discuss model binding in chapter 6.

ASP.NET Core can bind two different targets in Razor Pages:

- *Method arguments*—If a page handler has method arguments, the values from the request are used to create the required parameters.
- *Properties marked with a* `[BindProperty]` *attribute*—Any properties marked with the attribute will be bound. By default, this attribute does nothing for `GET` requests.

Model-bound values can be simple types, such as strings and integers, or they can be a complex type, as shown in the following listing. If any of the values provided in the request are *not* bound to a property or page handler argument, the additional values will go unused.

Listing 4.3 Example Razor Page handlers

```
public class SearchModel : PageModel
{
    private readonly SearchService _searchService;
    public SearchModel(SearchService searchService)
    {
        _searchService = searchService;
    }

    [BindProperty]
    public BindingModel Input { get; set; }
    public List<Product> Results { get; set; }

    public void OnGet()
    {
    }

    public IActionResult OnPost(int max)
    {
```

The SearchService is provided to the SearchModel for use in page handlers.

Undecorated properties will not be model-bound.

Properties decorated with the [BindProperty] attribute will be model-bound.

The page handler doesn't need to check if the model is valid. Returning void will render the view.

The max parameter in this page handler will be model-bound using the values in the request.

If the request was not valid, the method indicates the user should be redirected to the Index page.

```
        if (ModelState.IsValid)
        {
            Results = _searchService.Search (Input.SearchTerm, max);
            return Page();
        }
        return RedirectToPage("./Index");
    }
}
```

In this example, the `OnGet` handler doesn't require any parameters, and the method is simple—it returns `void`, which means the associated Razor view will be rendered. It could also have returned a `PageResult`; the effect would have been the same. Note that this handler is for HTTP `GET` requests, so the `Input` property decorated with `[BindProperty]` is not bound.

> **TIP** To bind properties for `GET` requests too, use the `SupportsGet` property of the attribute; for example, `[BindProperty(SupportsGet = true)]`.

The `OnPost` handler, conversely, accepts a parameter `max` as an argument. In this case it's a simple type, `int`, but it could also be a complex object. Additionally, as this handler corresponds to an HTTP `POST` request, the `Input` property is also model-bound to the request.

> **NOTE** Unlike most .NET classes, you can't use method overloading to have multiple page handlers on a Razor Page with the same name.

When an action method uses model-bound properties or parameters, it should always check that the provided model is valid using `ModelState.IsValid`. The `ModelState` property is exposed as a property on the base `PageModel` class and can be used to check that all the bound properties and parameters are valid. You'll see how the process works in chapter 6 when you learn about validation.

Once a page handler establishes that the method parameters provided to an action are valid, it can execute the appropriate business logic and handle the request. In the case of the `OnPost` handler, this involves calling the provided `SearchService` and setting the result on the `Results` property. Finally, the handler returns a `PageResult` by calling the base method

```
return Page();
```

If the model wasn't valid, you don't have any results to display! In this example, the action returns a `RedirectToPageResult` using the `RedirectToPage` helper method. When executed, this result will send a 302 redirect response to the user, which will cause their browser to navigate to the `Index` Razor Page.

Note that the `OnGet` method returns `void` in the method signature, whereas the `OnPost` method returns an `IActionResult`. This is required in the `OnPost` method in order to allow the C# to compile (as the `Page` and `RedirectToPage` helper methods

return different types), but it doesn't change the final behavior of the methods. You could just as easily have called `Page` in the `OnGet` method and returned an `IActionResult`, and the behavior would be identical.

> **TIP** If you're returning more than one type of result from a page handler, you'll need to ensure your method returns an `IActionResult`.

In the next section we'll look in more depth at action results and what they're used for.

4.3.2 Returning responses with ActionResults

In the previous section, I emphasized that page handlers decide what *type* of response to return, but they don't generate the response themselves. It's the `IActionResult` returned by a page handler that, when executed by the Razor Pages infrastructure using the view engine, will generate the response.

This approach is key to following the MVC design pattern. It separates the decision of what sort of response to send from the generation of the response. This allows you to easily test your action method logic to confirm that the right sort of response is sent for a given input. You can then separately test that a given `IActionResult` generates the expected HTML, for example.

ASP.NET Core has many different types of `IActionResult`:

- `PageResult`—Generates an HTML view for an associated page in Razor Pages
- `ViewResult`—Generates an HTML view for a given Razor view when using MVC controllers
- `RedirectToPageResult`—Sends a 302 HTTP redirect response to automatically send a user to another page
- `RedirectResult`—Sends a 302 HTTP redirect response to automatically send a user to a specified URL (doesn't have to be a Razor Page)
- `FileResult`—Returns a file as the response
- `ContentResult`—Returns a provided string as the response
- `StatusCodeResult`—Sends a raw HTTP status code as the response, optionally with associated response body content
- `NotFoundResult`—Sends a raw 404 HTTP status code as the response

Each of these, when executed by Razor Pages, will generate a response to send back through the middleware pipeline and out to the user.

> **TIP** When you're using Razor Pages, you generally won't use some of these action results, such as `ContentResult` and `StatusCodeResult`. It's good to be aware of them, though, as you will likely use them if you are building Web APIs with MVC controllers.

In this section I'll give a brief description of the most common `IActionResult` classes that you'll use with Razor Pages.

PAGERESULT AND REDIRECTTOPAGERESULT

When you're building a traditional web application with Razor Pages, usually you'll be using `PageResult`, which generates an HTML response using Razor. We'll look at how this happens in detail in chapter 7.

You'll also commonly use the various redirect-based results to send the user to a new web page. For example, when you place an order on an e-commerce website, you typically navigate through multiple pages, as shown in figure 4.12. The web application sends HTTP redirects whenever it needs you to move to a different page, such as when a user submits a form. Your browser automatically follows the redirect requests, creating a seamless flow through the checkout process.

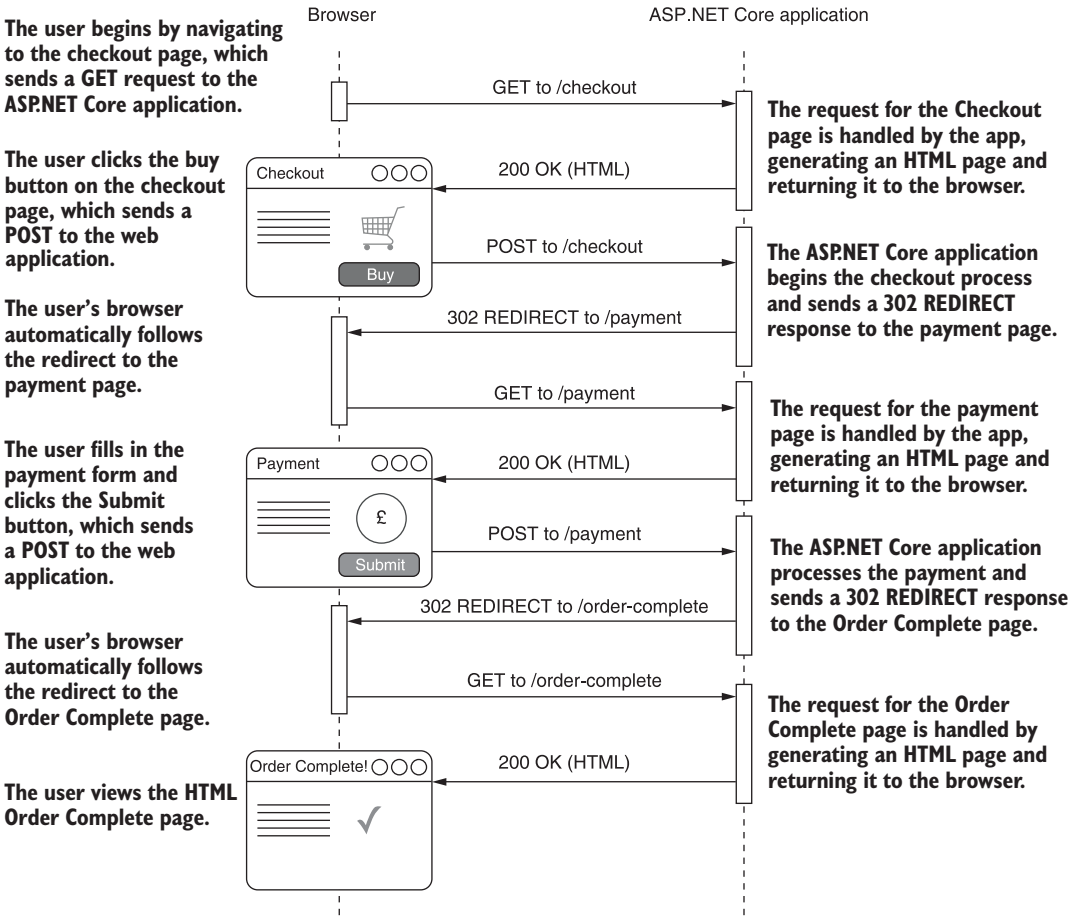

Figure 4.12 A typical `POST`, `REDIRECT`, `GET` flow through a website. A user sends their shopping basket to a checkout page, which validates its contents and redirects to a payment page without the user having to manually change the URL.

In this flow, whenever you return HTML you use a `PageResult`; when you redirect to a new page, you use a `RedirectToPageResult`.

> **TIP** Razor Pages are generally designed to be stateless, so if you want to persist data between multiple pages, you need to place it in a database or similar store. If you just want to store data for a single request, you may be able to use `TempData`, which stores small amounts of data in cookies for a single request. See the documentation for details: http://mng.bz/XdXp.

NotFoundResult and StatusCodeResult

As well as HTML and redirect responses, you'll occasionally need to send specific HTTP status codes. If you request a page for viewing a product on an e-commerce application, and that product doesn't exist, a 404 HTTP status code is returned to the browser, and you'll typically see a "Not found" web page. Razor Pages can achieve this behavior by returning a `NotFoundResult`, which will return a raw 404 HTTP status code. You could achieve a similar result using `StatusCodeResult` and setting the status code returned explicitly to 404.

Note that `NotFoundResult` doesn't generate any HTML; it only generates a raw 404 status code and returns it through the middleware pipeline. But, as discussed in the previous chapter, you can use the `StatusCodePagesMiddleware` to intercept this raw 404 status code after it's been generated and provide a user-friendly HTML response for it.

Creating ActionResult classes using helper methods

`ActionResult` classes can be created and returned using the normal `new` syntax of C#:

```
return new PageResult()
```

However, the Razor Pages `PageModel` base class also provides a number of helper methods for generating responses. It's common to use the `Page` method to generate an appropriate `PageResult`, the `RedirectToPage` method to generate a `RedirectToPageResult`, or the `NotFound` method to generate a `NotFoundResult`.

> **TIP** Most `ActionResult` classes have a helper method on the base `PageModel` class. They're typically named `Type`, and the result generated is called `TypeResult`. For example, the `StatusCode` method returns a `StatusCodeResult` instance.

As discussed earlier, the act of *returning* an `IActionResult` doesn't immediately generate the response—it's the *execution* of an `IActionResult` by the Razor Pages infrastructure, which occurs outside the action method. After producing the response, Razor Pages returns it to the middleware pipeline. From there, it passes through all the registered middleware in the pipeline, before the ASP.NET Core web server finally sends it to the user.

By now, you should have an overall understanding of the MVC design pattern and how it relates to ASP.NET Core and Razor Pages. The page handler methods on a Razor Page are invoked in response to given requests and are used to select the type of response to generate by returning an `IActionResult`.

It's important to remember that the MVC and Razor Pages infrastructure in ASP.NET Core runs as part of the `EndpointMiddleware` pipeline, as you saw in the previous chapter. Any response generated, whether a `PageResult` or a `RedirectToPageResult`, will pass back through the middleware pipeline, providing a potential opportunity for middleware to observe the response before the web server sends it to the user.

An aspect I've only vaguely touched on is how the `RoutingMiddleware` decides which Razor Page and handler to invoke for a given request. You don't want to have a Razor Page for *every* URL in an app. It would be difficult to have, for example, a different page per product in an e-shop—every product would need its own Razor Page! Handling this and other scenarios is the role of the routing infrastructure, and it's a key part of ASP.NET Core. In the next chapter, you'll see how to define routes, how to add constraints to your routes, and how they deconstruct URLs to match a single Razor Page handler.

Summary

- The MVC design pattern allows for a separation of concerns between the business logic of your application, the data that's passed around, and the display of data in a response.
- Razor Pages are built on the ASP.NET Core MVC framework, and they use many of the same primitives. They use conventions and a different project layout to optimize for page-based scenarios.
- MVC *controllers* contain multiple *action methods,* typically grouped around a high-level entity. Razor Pages groups all the *page handlers* for a single page in one place, grouping around a page/feature instead of an entity.
- Each Razor Page is equivalent to a mini controller focused on a single page, and each Razor Page handler corresponds to a separate action method.
- Razor Pages should inherit from the `PageModel` base class.
- A single Razor Page handler is selected based on the incoming request's URL, the HTTP verb, and the request's query string, in a process called *routing*.
- Page handlers should generally delegate to services to handle the business logic required by a request, instead of performing the changes themselves. This ensures a clean separation of concerns that aids testing and improves application structure.
- Page handlers can have parameters whose values are taken from properties of the incoming request in a process called *model binding*. Properties decorated with `[BindProperty]` can also be bound to the request.

- By default, properties decorated with `[BindProperty]` are not bound for `GET` requests. To enable binding, use `[BindProperty(SupportsGet = true)]`.
- Page handlers can return a `PageResult` or `void` to generate an HTML response.
- You can send users to a new Razor Page using a `RedirectToPageResult`.
- The `PageModel` base class exposes many helper methods for creating an `ActionResult`.